組込みシステムにおけるスケジューリングテーブル作成法 (最適化モデルとアルゴリズムの新展開)

AUTHOR(S):

杉山, 太一郎; 田中, 勇真; 橋本, 英樹; 柳浦, 睦憲

CITATION:

杉山, 太一郎 ...[et al]. 組込みシステムにおけるスケジューリングテーブル作成法 (最適化モデルとアルゴリズムの新展開). 数理解析研究所講究録 2011, 1726: 199-213

ISSUE DATE:

2011-02

URL:

http://hdl.handle.net/2433/170496

数理解析研究所講究録
第1726巻 2011年 199-213

組込みシステムにおける スケジューリングテーブル作成法

名古屋大学大学院情報学研究科　杉山 太一郎 (Taichiro Sugiyama)
Graduate School of Information Science, Nagoya University
名古屋大学大学院情報学研究科　田中 勇真 (Yuma Tanaka)
Graduate School of Information Science, Nagoya University
中央大学理工学部　橋本 英樹 (Hideki Hashimoto)
Faculty of Science and Engineering, Chuo University
名古屋大学大学院情報科学研究科　柳浦 睦憲 (Mutsunori Yagiura)
Graduate School of Information Science, Nagoya University

1　はじめに

リアルタイムシステムは, 携帯電話や情報家電のようにリアルタイムに入力を受けつけ, その入力に対する出力をある時間内に行うようなシステムの総称である [5]. その動作はタスクという繰り返し実行されるプロセスの実行単位によって行われる. リアルタイムシステムには時間制約が存在するので, その時間制約を守るようにタスクの実行順序を定める必要があり, その実行順序を定めることをタスクスケジューリングという.

代表的なスケジューリング手法として, 1970年代にLiuとLayland [3] によって解析された「固定優先度スケジューリング」がある. 固定優先度スケジューリングとは, 各タスクに割り当てられた周期と優先度に基づいて各タスクを周期的に実行しスケジューリングを行う手法であり, 現在一般に用いられている. しかし, 固定優先度スケジューリングにはCPUの使用率などの観点から限界があり [5], 新たなスケジューリング方法の需要が出てきた. そこで, 本研究では「スケジューリングテーブルに基づく方法」を扱う. スケジューリングテーブルに基づく方法とは, ある時間区間のスケジュールをスケジューリングテーブルと呼ぶことにして, それを繰り返し実行することにより, スケジューリングを行う手法である.

リアルタイムシステムのスケジューリングにおいて考えないといけないものにシステムの応答時間がある. リアルタイムシステムには様々な種類の入力があり, 各入力は対応するいくつかのタスクによる処理が所定の順に行われた後出力に至る. 入力された情報を所定の順に処理して出力に至るまでに必要なタスクの並びをパスと呼ぶ. 各パスには許容遅れの値が設定されており, 任意の時刻に発生し得る入力からそれに対する出力までの時間を許容遅れ以下に抑える必要がある. 固定優先度スケジューリングに対しては, 文献 [1] で, 許容遅れをできるだけ満たすような固定優先度スケジューリングの周期と優先度を求め

るアルゴリズムが提案されている. 文献 [4] では, 将来的な分散システムを対象として現実的な問題設定と, アニーリング法 (SA 法) により各タスクの優先度や周期を探索し, それらに基づきスケジューリングを行う手法が提案されている.

本研究では, 中断不可能なタスクのみから構成されるリアルタイムシステムに対して, 応答時間に対する許容遅れ違反度が最小になるようなスケジューリングテーブルを得る手法を提案する. スケジューリングテーブルの探索法として, 単純な局所探索法と, 与えられたスケジュール上のタスクに対し先行関係を考慮することで高速化を図った局所探索法の 2 つを提案する. 計算実験により, これらの効果を確認する.

2 モデル化

本節では, タスクやパスなどを定義し, リアルタイムシステムのモデル化を行う.

2.1 タスクとパスの定義

$\mathcal{T} = \{1, 2, \ldots, n\}$ をタスクの集合とする. $\mathcal{T}$ は中断不可能なタスク集合から構成され, それぞれのタスク $i \in \mathcal{T}$ は処理時間 $c_i (> 0)$ を持つ. 中断不可能なタスクとは, タスクの実行を中断できないものをいう.

$\mathcal{P} = \{1, 2, \ldots, m\}$ をパスの集合とする. 各パスはタスクの並びである. パス $p \in \mathcal{P}$ に含まれるタスクの数を l_p とし, $q_p(h)$ をパス p の h 番目のタスクとする. それぞれのパス $p \in \mathcal{P}$ は時間制約として許容遅れ $\theta_p (> 0)$ を持つ. ある時刻の入力がパス p によって処理されるとは, そのデータがタスク $q_p(1), q_p(2), \ldots, q_p(l_p)$ によってこの順に処理されることを言う. その際, パス $p \in \mathcal{P}$ の h 番目のタスク $q_p(h)$ は, 実行時に直前の $q_p(h-1)$ の実行結果を参照して実行される. また, リアルタイムシステムへの任意の時刻の入力に対して, パス p は入力された時刻から θ_p 以内にその入力を処理し出力を行わなければならない.

図 1 に 3 つのタスク (タスク A, B ,C) のスケジュールの例を示した. 各タスクの実行をタスクの種類ごとに 3 行に分けて表示している. タスク B と C をこの順に行うパス p に注目し, この実行において, 下流タスク $q_p(h)$ が上流タスク $q_p(h-1)$ の実行結果を参照しているペアを黒線で結んだ.

2.2 システムの応答時間

本節では, 与えられたスケジュールを効率的に評価するためのいくつかの概念を説明する.

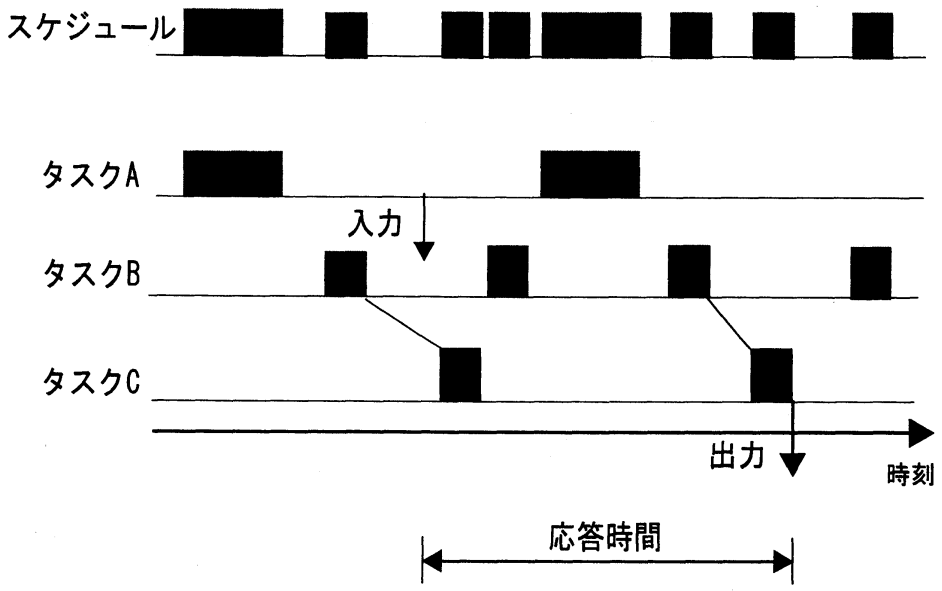

図 1: スケジュールの例

2.2.1 有効なパス

あるスケジュールにおいて, タスク $i \in \mathcal{T}$ の j 回目の実行を π_i^j, π_i^j の開始時刻を s_i^j, 終了時刻を e_i^j とおく. また, スケジュール上のタスク i の実行回数を b_i とする. このスケジュールで, ある時刻の入力がパス p によって処理されることを考え, その処理を行う各タスクの実行を

$$(\pi_{q_p(1)}^{j_1}, \pi_{q_p(2)}^{j_2}, \ldots, \pi_{q_p(l_p)}^{j_{l_p}})$$

とおく. ここで, $j_h (h = 1, \ldots, l_p - 1)$ に対して, タスク $q_p(h+1)$ はタスク $q_p(h)$ の直前の実行結果を参照することから,

$$e_{q_p(h)}^{j_h} \leq s_{q_p(h+1)}^{j_{h+1}} \tag{1}$$

であり,

$$j_h = \arg \max_{k: e_{q_p(h)}^k \leq s_{q_p(h+1)}^{j_{h+1}}} e_{q_p(h)}^k \tag{2}$$

が成り立つ. さらに $\pi_{q_p(1)}^{j_1}$ で処理された結果を初めて処理する $q_p(l_p)$ の実行が $\pi_{q_p(l_p)}^{j_{l_p}}$ であるとき, このタスクの実行 $(\pi_{q_p(1)}^{j_1}, \pi_{q_p(2)}^{j_2}, \ldots, \pi_{q_p(l_p)}^{j_{l_p}})$ を有効なパスと呼ぶ. 与えられたスケジュールにおいて, パス p の有効なパス全てをそれらの最後のタスクの実行開始時刻の昇順に並べたとき, 前から l 番目のものをパス p の l 本目の有効なパスという. また, パス p の l 本目の有効なパスを構成するタスクの順序列を

$$\phi_l^p = (\pi_{q_p(1)}^{j_1}, \pi_{q_p(2)}^{j_2}, \ldots, \pi_{q_p(l_p)}^{j_{l_p}})$$

Algorithm 1 パス p の有効なパスを検出するアルゴリズム

1: $j_k := 1, \forall k = 1, 2, \ldots, l_p$
2: $Y := \emptyset$ //パス p の有効なパスの集合
3: **while** $j_{l_p} \leq b_{q_p(l_p)}$ **do**
4: $h := l_p - 1$
5: **while** $h > 0$ **do**
6: $j_h := \arg\max_{k:e^k_{q_p(h)} \leq s^{j_{h+1}}_{q_p(h+1)}} e^k_{q_p(h)}$
7: $h := h - 1$
8: **end while**
9: **if** Y に最後に追加した有効なパスの先頭が $\pi^{j_1}_{q_p(1)}$ ではない **then**
10: $Y := Y \cup (\pi^{j_1}_{q_p(1)}, \pi^{j_2}_{q_p(2)}, \ldots, \pi^{j_{l_p}}_{q_p(l_p)})$
11: **end if**
12: $j_{l_p} = j_{l_p} + 1$
13: **end while**

と表し,

$$\phi^p_l(h) = \pi^{j_h}_{q_p(h)}$$

とする. 与えられたスケジュールにおいて, パス p の有効なパスの本数を T_p とする.

有効なパスの例を図 2 に示した. 矢印でつながれたパスが有効なパスである.

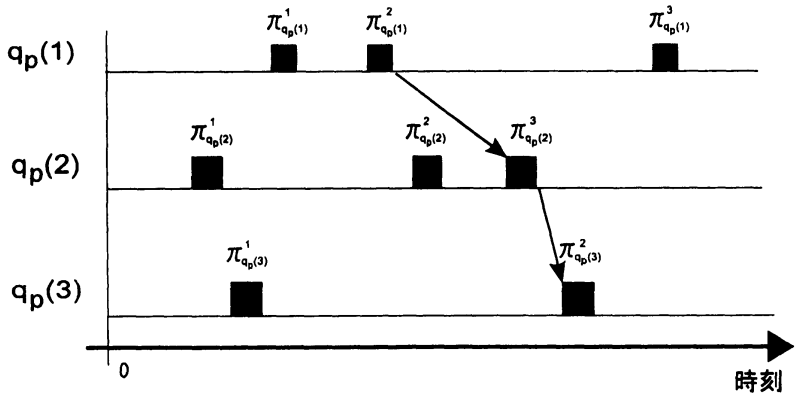

図 2: 有効なパスの例 (パス p の 1 本目の有効なパス $\phi^p_1 = (\pi^2_{q_p(1)}, \pi^3_{q_p(2)}, \pi^2_{q_p(3)})$)

アルゴリズム 1 にパス p の有効なパスを求めるアルゴリズムを示した. アルゴリズムはスケジュール上の各 $q_p(l_p)$ の実行に対して, 式 (2) を満たす j_h を $h = l_p - 1, l_p - 2, \ldots, 1$ の順に求めることで, スケジュール上のパス p のすべての有効なパスを見つける.

アルゴリズム1の計算時間を考える．ステップ6では，$e^k_{q_p(h)} \leq s^{j_{h+1}}_{q_p(h+1)}$ の範囲で，$e^k_{q_p(h)}$ が最大になる k を求めている．すべての $q_{p(h)}$ の実行から，そのような j_h を求めると，ステップ6のたびに，$\mathrm{O}\left(\sum_{i\in\mathcal{T}} b_i\right)$ 時間かかる．しかし，スケジュール上のタスク i の実行を実行終了時刻の小さい順にリスト A_i として用意し，$q_{p(h)}$ の各実行を $A_{q_{p(h)}}$ の中から終了時刻の小さい順に調べていき，以前調べたものより終了時刻が小さいものを調べないようにすると，計算時間を短縮できる．ここで，すべてのタスク i の A_i を用意するのにかかる計算時間は，テーブル上のすべてのタスクを走査する必要があるため，$\mathrm{O}\left(\sum_{i\in\mathcal{T}} b_i\right)$ である．ステップ6での $q_{p(h+1)}$ の実行開始時刻 $s^{j_{h+1}}_{q_p(h+1)}$ は，アルゴリズムの進行と共に大きくなる．そのため，以前調べた $q_{p(h)}$ の終了時刻より小さい終了時刻を持つタスクの実行を，有効なパスに含まれるタスクの実行として選ぶことはない．従って，j_h の計算で $q_{p(h)}$ の各タスクの実行の終了時刻を $A_{q_{p(h)}}$ から調べる際に，以前に調べた $q_{p(h)}$ の終了時刻より，終了時刻が小さい $q_{p(h)}$ の実行を調べる対象からはずすことができる．そのため，アルゴリズム全体において，パス p を構成する各タスクの実行はそれぞれ一度ずつしか参照されない．従って，ステップ6での総計算時間は，$\mathrm{O}\left(\sum_{h=1}^{l_p} b_{q_p(h)}\right)$ となる．また，その他のステップについてはすべて定数時間で計算することができるため，アルゴリズム1の総計算時刻は，$\mathrm{O}\left(\sum_{h=1}^{l_p} b_{q_p(h)}\right)$ となる．ただし，上述したように前処理として，各タスク i に対して A_i を用意するのに全体で $\mathrm{O}\left(\sum_{i\in\mathcal{T}} b_i\right)$ 時間かかる．

2.2.2 応答時間

与えられたスケジュールにおいて一つの有効なパスがどの時間区間の入力に対して応答できるのかを考える．パス p の l 本目の有効なパス ϕ^p_l の開始時刻と終了時刻をそれぞれ $t^p_{\mathrm{start}}(l)$，$t^p_{\mathrm{end}}(l)$ とする．このとき，時刻 t の入力をパス p が処理することを考える．$t^p_{\mathrm{start}}(l-1) < t \leq t^p_{\mathrm{start}}(l)$ とすると，出力は時刻 $t^p_{\mathrm{end}}(l)$ に得られる．従って応答時間は $t^p_{\mathrm{end}}(l) - t$ となるため，最悪の場合は

$$\lim_{\varepsilon\to+0}\{t^p_{\mathrm{end}}(l) - (t^p_{\mathrm{start}}(l-1)+\varepsilon)\} = t^p_{\mathrm{end}}(l) - t^p_{\mathrm{start}}(l-1)$$

となる．従ってパス p の2本の隣り合う有効なパスの開始時刻と終了時刻の差 $t^p_{\mathrm{end}}(l) - t^p_{\mathrm{start}}(l-1)$ が応答時間となる．また，この値をパス p の l 本目の有効なパスの応答時間 d^p_l と呼ぶ．与えられたスケジュールでの，パス p に対する最大応答時間 r_p は，

$$r_p = \max_l d^p_l$$

となる．これを用いてリアルタイムシステムの許容遅れに対する条件は

$$r^p \leq \theta^p, \quad p \in \mathcal{P}$$

と表せる．

3 スケジューリング

この節では, 本研究で設計するスケジューリングテーブルに基づくスケジューリング方法を紹介する. また, 提案するアルゴリズムの中で初期スケジューリングテーブルを求めるのに使う, 固定優先度スケジューリングによるスケジューリング方法についても述べる.

3.1 固定優先度スケジューリング

優先度に基づく周期的スケジューリングは, 各タスクに対する周期とタスク間の優先度を定めるルールが与えられ, それらに基づきスケジューリングを行う方法である. 各タスクは割り当てられた周期で起動され, 起動されているものの中で最も優先度が高いタスクが実行される. 特に, スケジュールしている間に各タスクに割り当てられた優先度が変更されることのないものを固定優先度スケジューリングという.

図3に, 固定優先度スケジューリングの例を示した. タスクの優先度はA, B, Cの順に高いとし, 図において上向きの矢印はタスクの起動を表す. 図3において, タスクB, Cが同時に起動しているタイミングがあるが, タスクBのほうがタスクCより優先度が高いため, まずタスクBが実行され, その後にタスクCが実行されている.

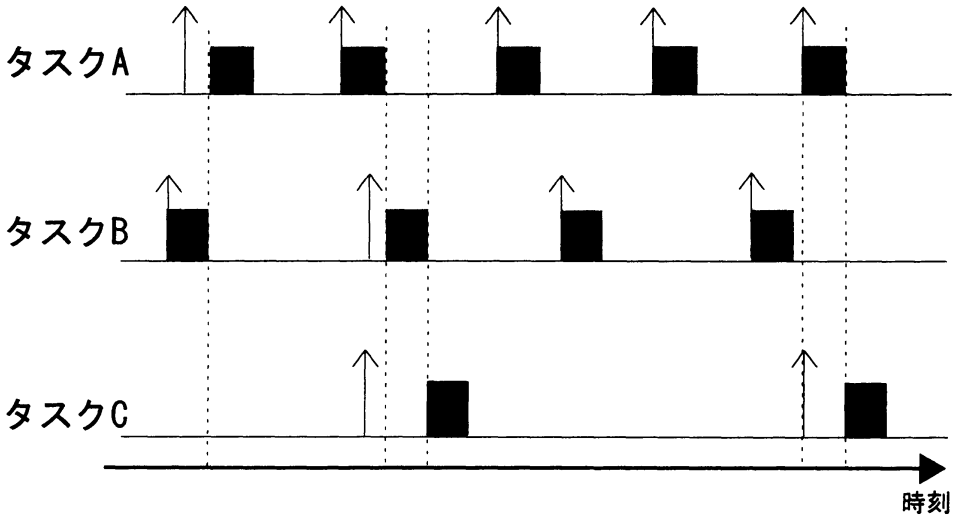

図 3: 固定優先度スケジューリングの例

3.2 スケジューリングテーブルに基づくスケジューリング

スケジューリングテーブルに基づく方法とは, ある時間区間のスケジュールをスケジューリングテーブルと呼ぶことにして, それを繰り返し実行することにより, スケジューリングを行う手法である. 本研究では, スケジューリングテーブル上のタスクの実行がない区間をすべて前詰めにしたものを考える. このようにタスクの実行がない区間を前詰めにしても, すべての応答時間は大きくならないからである.

テーブル上のタスクの総実行回数を L としたとき, スケジューリングテーブル上のタスクの実行の列を σ とし, k 番目 $(k = 1, 2, \ldots, L)$ のタスク実行を $\sigma(k)(\in \{\pi_i^j \mid j = 1, 2, \ldots, b_i, i \in \mathcal{T}\})$ とする. 図 4 に, スケジューリングテーブルに基づくスケジューリングの例を示した. 図では与えられたスケジューリングテーブルを繰り返し実行しスケジューリングを行う様子が示されている.

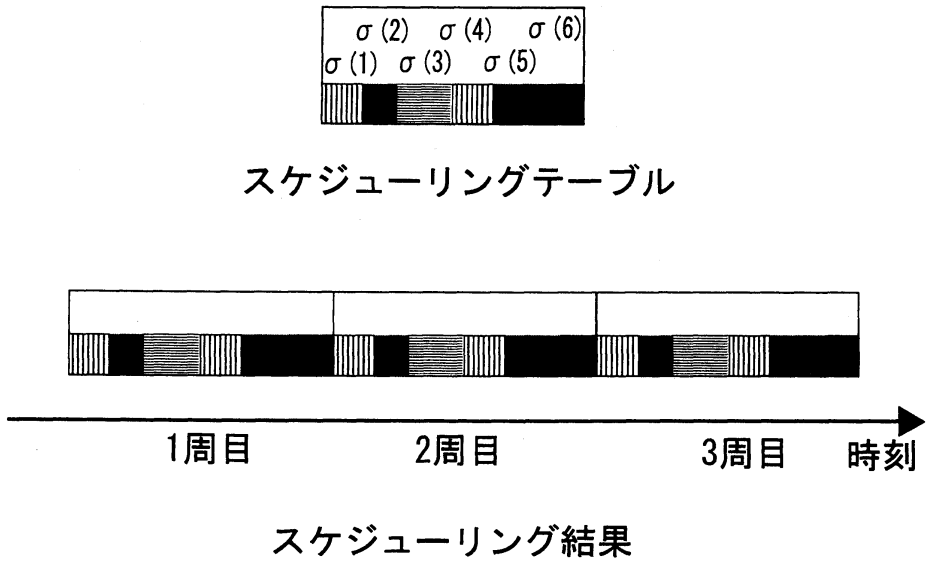

図 4: スケジューリングテーブルに基づくスケジューリングの例

3.3 スケジュールの評価方法

スケジュールの評価には, 以下の 3 つの評価値を用いる. $(x)_+ = \max\{x, 0\}$ としたとき,

$$
\begin{aligned}
f_1 &= \max_{p\in\mathcal{P}} \max_{i=1,\ldots,T_p-1} \frac{(d_i^p-\theta^p)_+}{\theta^p} \\
f_2 &= \sum_{p\in\mathcal{P}} \max_{i=1,\ldots,T_p-1} \frac{(d_i^p-\theta^p)_+}{\theta^p} \\
f_3 &= \sum_{p\in\mathcal{P}} \sum_{i=1}^{T_p-1} \frac{(d_i^p-\theta^p)_+}{\theta^p}
\end{aligned}
$$

とする. f_1 はすべての応答時間の中で許容遅れに対する違反度が最も大きなもの, f_2 は各パスの応答時間の中で違反度が最も大きなのものの総和, f_3 はすべての応答時間の違反度の総和である. スケジューリングテーブル σ の f_1, f_2, f_3 の値をそれぞれ $f_1(\sigma), f_2(\sigma), f_3(\sigma)$ と表す. 2つのテーブルは評価値 (f_1, f_2, f_3) の辞書式順序で比較される. 辞書式順序で評価するとは, 2つのスケジューリングテーブル σ と σ' を比較するとき, $f_1(\sigma) < f_1(\sigma')$ ならば σ の方が良い解, $f_1(\sigma) = f_1(\sigma')$ のときには $f_2(\sigma) < f_2(\sigma')$ ならば σ の方が良い解, $f_1(\sigma) = f_1(\sigma')$ かつ $f_2(\sigma) = f_2(\sigma')$ のときには $f_3(\sigma) < f_3(\sigma')$ ならば σ の方が良い解であると評価することをいう. このとき, σ と σ' を比べて, σ が σ' より良い解であるとき, $f(\sigma) < f(\sigma')$ と表す.

4 アルゴリズム

本研究では, スケジューリングテーブル σ を局所探索法によって求める. すなわち, 現在の σ より良い解が近傍 $N(\sigma)$ 内にあればそれに置き換える, という操作を反復し解を改善していく.

4.1 初期解

局所探索法の初期解としてのスケジューリングテーブルを得るために, 固定優先度スケジューリングをある時間だけシミュレートする. 文献 [1] で提案されているアルゴリズムによりリアルタイムシステムの許容遅れをできるだけ満たす固定優先度スケジューリングの優先度と周期が求まる. シミュレータの入力は, タスク集合と得られた優先度と周期と各タスクの起動オフセットである. 起動オフセットとは, タスクを起動する最初の起動時刻のことで, 計算実験ではすべて0とした.

次に, 有効なパスに含まれないタスクの実行をスケジュールから削除しても, テーブル上の有効なパスは変化せず, すべての応答時間は変化しないため, どの有効なパスにも含まれないタスクの実行を削除する. 不要なタスクの削除は, 有効なパスに含まれる各タスクの実行にチェックをし, そのチェックが一つも

ついていないタスクをシミュレーション結果から削除する．そのため，計算時間として有効なパスを計算するのにかかる時間を必要とする．従って，不要なタスクの削除にかかる計算時間は，$\mathrm{O}\left(\sum_{p\in\mathcal{P}}\sum_{h=1}^{l_p} b_{q_p(h)}\right)$ となる．

最後にテーブル上のタスクの実行が無い区間を前詰めにすることで初期解 σ_{init} を得る．

4.2 近傍

本研究では近傍 $N(\sigma)$ としてシフト近傍を用いる．シフト近傍 N_{shift} は現在の解 σ の一つの要素を σ の他の隣り合う要素と要素の間にシフトすることで得られる解集合のことである．例えば，$\sigma=(\pi_2^1,\pi_1^1,\pi_3^1,\pi_4^1,\pi_1^2)$ であったときに，σ の π_2^1 を π_3^1 と π_4^1 の間にシフトすると $\sigma=(\pi_1^1,\pi_3^1,\pi_2^1,\pi_4^1,\pi_1^2)$ となる．シフト近傍のサイズは L 個の要素に対してシフト候補がそれぞれ $L-1$ 個あるため，$\mathrm{O}(L^2)$ となる．

4.3 先行関係用いた局所探索

スケジューリングテーブル上の各タスクの実行に現在の有効なパスを壊さないような先行関係を付与する．この先行関係を用い，テーブル上の有効なパスを保存する近傍解に限定して探索を行う．これにより，近傍操作を行っても有効なパスが保存されるため，近傍操作のたびに有効なパスの検出を行う必要がなくなるため，計算時間の短縮が可能となる．

4.3.1 先行関係の付与

スケジュール上のタスクの実行を頂点としたグラフ $G=(V,E)$ を考え，スケジュール上の有効なパスを保存するようなタスクの実行同士の時間的な順序関係を辺で表す．この順序関係のことを先行関係と呼ぶ．このとき，V は，

$$V=\{\sigma(k)\mid k=1,2,\ldots,L\}$$

となる．E は，

$$E=\{(\phi_l^p(h),\phi_{l+1}^p(h-1))\mid p\in\mathcal{P},l=1,2,\ldots,T_p-1,h=2,3,\ldots,l_p\}\cup$$

$$\{(\phi_l^p(h),\phi_l^p(h+1))\mid p\in\mathcal{P},l=1,2,\ldots,T_p,h=1,2,\ldots,l_p-1\}$$

とする．こうしてできた $G=(V,E)$ を先行関係のグラフと呼ぶ．

ここで，$\phi_l^p(h)$ から $\phi_{l+1}^p(h-1)$ に対して，有向辺があるのは以下の理由による．$\phi_l^p(h)$ が $\phi_{l+1}^p(h-1)$ をスケジューリングテーブル上で追い越すと，式 (2) より，$\phi_l^p(h)$ が参照するタスクの実行は，$\phi_{l+1}^p(h-1)$ またはそれより大きい開始時

刻を持つタスクの実行になる．これにより，$\phi_l^p(h)$ が参照していたタスクの実行が $\phi_l^p(h-1)$ から他のタスクの実行へと変わってしまい，スケジューリングテーブル上の有効なパスを保存できない．また，同様に $\phi_l^p(h)$ が $\phi_l^p(h+1)$ を追い越すと，$\phi_l^p(h+1)$ の参照先が $\phi_l^p(h)$ またはそれよりも小さい開始時刻をもつタスクの実行となるか，参照先がなくなるため，スケジューリングテーブル上の有効なパスを保存できない．

あるスケジュールに先行関係を付与した例を図5に示す．図5においてパスはタスクA, B, Cからなるものとし，先行関係を矢印で示した．

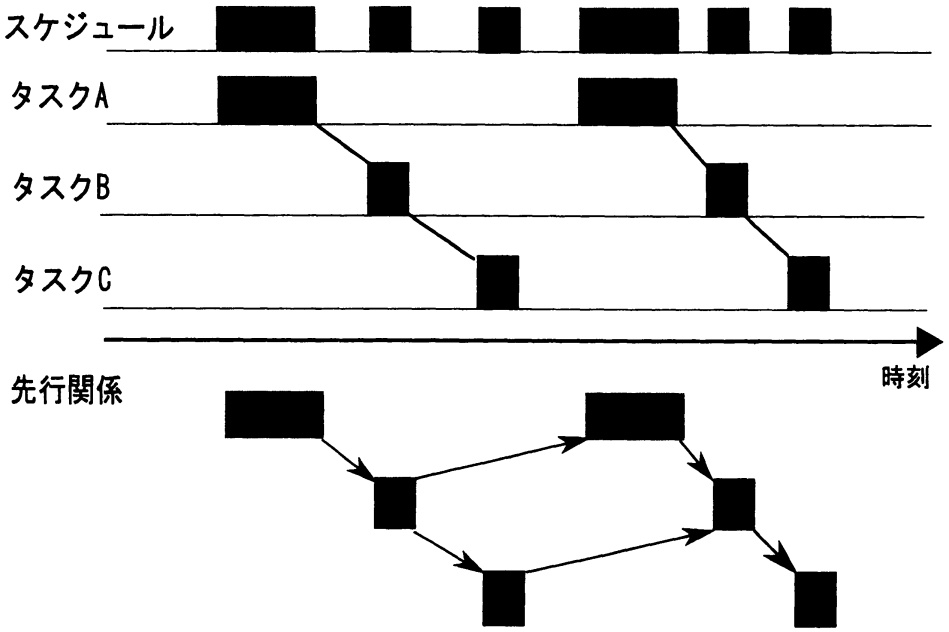

図 5: 先行関係の例

4.3.2 先行関係を考慮した近傍

有効なパスを壊さないタスクのシフト操作によって，パス p の l 本目の有効なパスの応答時間 d_l^p を改善することを考える．ここで $\sigma(k_1) = \phi_{l-1}^p(1)$, $\sigma(k_2) = \phi_l^p(l_p)$ となる k_1 と k_2 に対して，タスクの実行の部分列 $\sigma_{p,l}$ を $\sigma_{p,l}(k) = \sigma(k_1 + k - 1), k = 1, 2, \ldots, k_2 - k_1 + 1$ と定義し，便宜上区間と呼び，$\sigma_{p,l}(k) = v$ のとき $\sigma_{p,l}^{-1}(v) = k$ と表記する．また，タスクの実行を $\sigma_{p,l}$ の右端にシフトするとは，$\sigma(k), k = k_1, \ldots, k_2$ のタスクの実行一つを $\sigma(k_2)$ と $\sigma(k_2+1)$ の間に移動するということである．$d_l^p = t_{\mathrm{end}}^p(l) - t_{\mathrm{start}}^p(l-1)$ であるから，$\sigma_{p,l}$ の要素の中で，先行関係を守りつつ $\sigma_{p,l}$ 外へのシフトが可能なものを $\sigma_{p,l}$ 外にシフトすれば，d_l^p はそのようなタスク i の処理時間 c_i だけ小さくなる．図6に先行関係を保存したままタスクの実行を区間の右端にシフトすることにより応答時間が改善する例を示す．パス p_1 はタスクAとB，パス p_2 はタスクDとCからなるものとし

た. 図6において, タスクCの2回目の実行を$\sigma_{p_1,l}$の外にシフトする様子が示されている. このシフトにより, $d_l^{p_1}$がc_Cだけ減少しており, 先行関係および有効なパスは保存されている.

そこで, d_l^pを改善するために, 区間$\sigma_{p,l}$の要素で, 先行関係を保存したまま$\sigma_{p,l}$外にシフトすることで応答時間を改善することを考える. 区間内のすべてのタスクの実行に対し, 先行関係を保存するシフトがあるかどうか調べればよいが, 今回は区間内の先行関係を保存したままシフト可能なタスク実行のうち最も外側にあるものに限定した. これは, シフトする対象をこのように限定することで, 複数個のタスクの実行を同時にシフトする操作を考える必要がなくなり, 実装が容易になるためである. 例えば, 図6のタスクDの1回目の実行を区間の右外にシフトしようとすると, 先行関係を保存するためには, タスクCの2回目の実行も同時にシフトする必要が出てくる. 今回は実装しなかったが, 同時にシフトしなければならないタスクの検出は, 例えば区間内のタスクの実行の先行関係に対する推移閉包を計算することで可能で, 茨木ら [2] の方法を用いると$O((k_2-k_1)^3)$時間で計算できる.

与えられたテーブルの区間に対し, その中のタスクの実行の中で先行関係を保ちながら右端にシフト可能な最も右側のタスクを一つ見つけるアルゴリズムを, アルゴリズム2に示した. アルゴリズム2は, ある区間内の右から順に, 各タスクの実行に対して先行関係のグラフにおいて隣接する区間内のタスクの実行が右端にシフト可能かどうかをチェックする. もし, 隣接しているタスクに一つも右端にシフト不可能なものがなければ, 自身を右端にシフト可能なタスクとして出力して終了する. そうでなければ自身を右端にシフト不可能としてチェック対象を次のタスクに移動する.

アルゴリズム2は, 順序の区間$\sigma_{p,l}$に着目したときに, その区間内のタスクのうち, 有効なパスを保存したままその区間の右端にシフト可能なタスクを一つ見つけるか, もしくは存在しないということを出力する. なお, 左端にシフト可能かどうか調べる場合も同様の方法で可能である. また, σに対して区間$\sigma_{p,l}$に着目したとき, アルゴリズム2で見つかったタスクの実行を区間の右端にシフトすることにより得られるテーブルを$S(\sigma,p,l,\mathrm{R})$, 同様に左端にシフトすることにより得られるテーブルを$S(\sigma,p,l,\mathrm{L})$とする (RとLはそれぞれ右と左を表すフラグ).

この局所探索のアルゴリズムをアルゴリズム3に示した. この局所探索の近傍解においては, スケジュール上のすべての有効なパスのうち, 着目した$\sigma_{p,l}$にそのパスの始点もしくは終点であるようなタスクの実行を含むものの応答時間のみが増減する. 従って, スケジュール上の有効なパスの再検出を行わずに, そのような応答時間の増減だけを調べるだけで近傍解の評価を行うことができる.

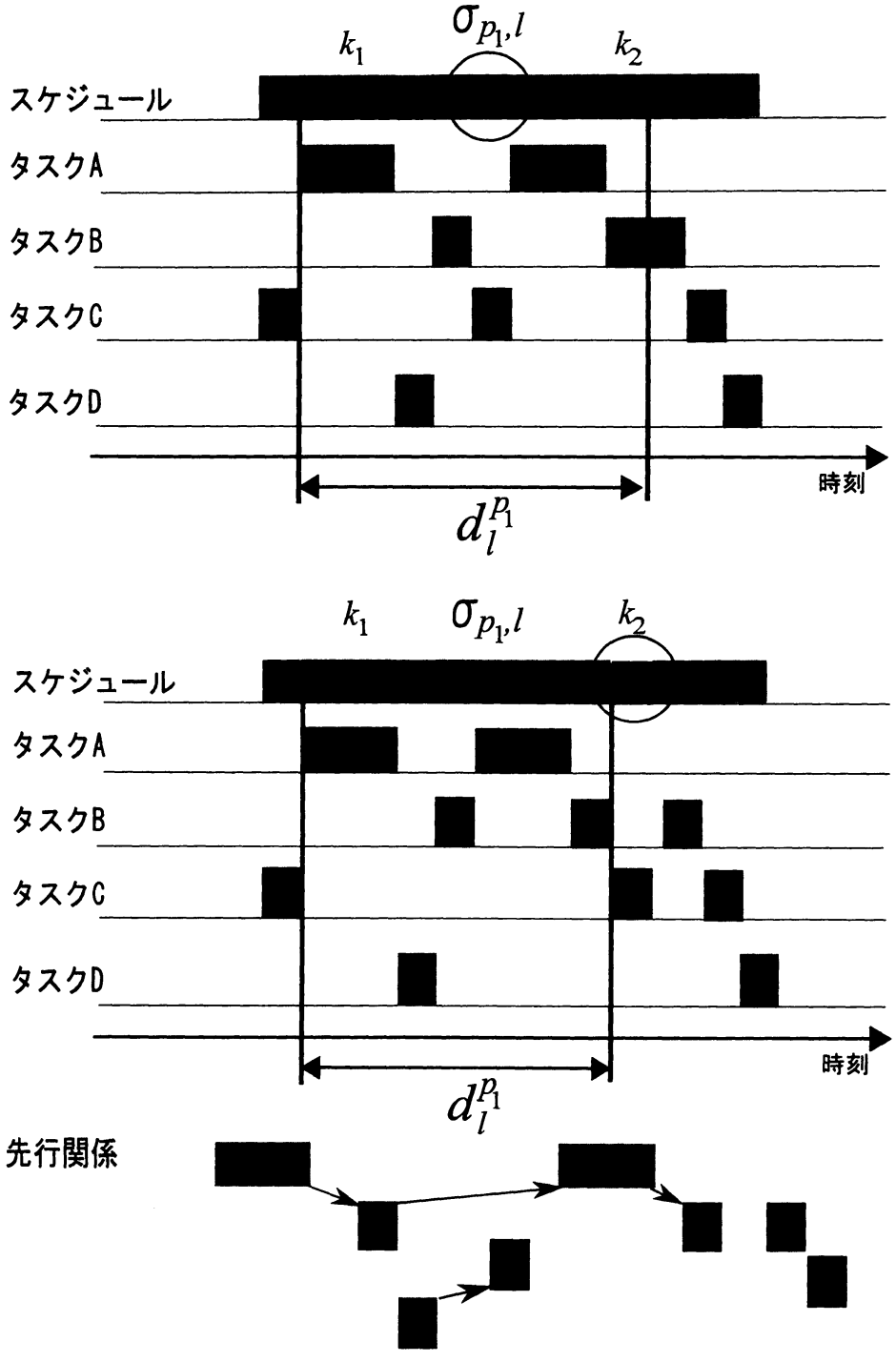

図 6: 先行関係を保存したままシフトすることにより応答時間が改善する例

Algorithm 2 $\sigma_{p,l}$ の要素で右端にシフト可能な最も右側のタスクを一つ見つけるアルゴリズム

1: $\sigma_{p,l}$ の要素数を $L_{\sigma_{p,l}}$ とする
2: 長さ $L_{\sigma_{p,l}}$ の配列 $mark$ を用意する　$//mark[k] = 1$ なら $\sigma_{p,l}(k)$ を右端にシフト可能, $mark[k] = -1$ なら右端にシフト不可能
3: $k := L_{\sigma_{p,l}}$
4: $mark[k] := -1$
5: **while** $k > 1$ **do**
6: 　$k := k - 1$, $mark[k] := 1$
7: 　$u = \sigma_{p,l}(k)$ から出る各辺 $(u, v) \in E$ に対して, v が $\sigma_{p,l}$ の要素の一つで, $mark[\sigma_{p,l}^{-1}(v)] = -1$ となるものが一つでも存在すれば, $mark[k] := -1$
8: 　**if** $mark[k] = 1$ **then**
9: 　　$\sigma_{p,l}(k)$ を出力して, アルゴリズムを終了する
10: 　**end if**
11: **end while**
12: $\sigma_{p,l}$ の要素で右端にシフト可能なタスクの実行はないことを出力してアルゴリズムを終了する

5 計算実験

考案したテーブル作成アルゴリズムの効果を見るため, 企業から提供された問題例に対して計算実験を行った. アルゴリズムを C 言語で実装し, 計算機は Core2Duo 2.40GHz, メモリ 2GB を用いた.

本研究で提案した単純な局所探索法と先行関係を用いた局所探索法をそれぞれ実行した. なお, 初期解には 4.1 節の方法で求めたものを使用した. 計算結果を表 1 と表 2 に示す. 表中, 初期解を σ_{init}, 局所探索の出力を σ_{opt} と記した. また, 計算時間が 4000 秒を超える問題例は 4000 秒で停止し, その時点での暫定解を出力し, 計算時間を表中で"$>$ 4000" と表した.

表 1: 単純な局所探索の結果

問題例	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{P}\rvert$	$f_1(\sigma_{\text{init}})$	$f_2(\sigma_{\text{init}})$	$f_3(\sigma_{\text{init}})$	$f_1(\sigma_{\text{opt}})$	$f_2(\sigma_{\text{opt}})$	$f_3(\sigma_{\text{opt}})$	時間
inst1	8	4	63	172	4,076	45	122	3,223	12.86
set_1-3	23	13	319	2,446	17,653	140	1,304	9,542	57.67
set_1-2	18	10	244	1,807	18,236	151	1,205	9,222	45.58
r200	200	2,942	27	9,854	21,992	15	3,933	5,923	$>$ 4000
r1000	1000	45,314	74	444,233	1,053,364	50	294,305	445,867	$>$ 4000

Algorithm 3 σ_{init} を初期解とした先行関係を用いた局所探索

1: $k := 1, k' := 1, k'' := 0$
2: $D :=$ R　//R(L) は右 (左) 方向へのシフトを行うことを表すフラグ
3: $P := \mathcal{P}$
4: $\sigma^{(k)} := \sigma_{init}$
5: **while do**
6:　　**while** $P \neq \emptyset$ **do**
7:　　$p \in P$ をランダムに一つ選ぶ
8:　　**for** $l = 1, \ldots, T_p - 1$ **do**
9:　　**if** $f(S(\sigma^{(k)}, p, l, D)) < f(\sigma^{(k)})$ **then**
10:　　$\sigma^{(k+1)} := S(\sigma^{(k)}, p, l, D)$
11:　　$k := k + 1$
12:　　break
13:　　**end if**
14:　　**end for**
15:　　$P := P \setminus \{p\}$
16:　　**end while**
17:　　**if** $k = k'$ **then**　//片方だけシフトで局所最適
18:　　**if** $k = k''$ **then**　//左右のシフトで改善なし
19:　　$\sigma^{(k)}$ を出力してアルゴリズム終了
20:　　**end if**
21:　　D =R なら D :=L とする, そうでないなら D :=R とする.
22:　　$k'' := k$
23:　　**end if**
24:　　$k' := k$
25:　　$P := \mathcal{P}$
26: **end while**

表 2: 先行関係を用いた局所探索の結果

問題例	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{P}\rvert$	$f_1(\sigma_{\text{init}})$	$f_2(\sigma_{\text{init}})$	$f_3(\sigma_{\text{init}})$	$f_1(\sigma_{\text{opt}})$	$f_2(\sigma_{\text{opt}})$	$f_3(\sigma_{\text{opt}})$	時間
inst1	8	4	63	172	4,076	57	137	916	0.00
set_1-3	23	13	319	2,446	17,653	259	1,793	12,093	0.02
set_1-2	18	10	244	1,807	18,236	186	1,284	14,874	0.03
r200	200	2,942	27	9,854	21,992	12	808	1,158	2.20
r1000	1000	45,314	74	444,233	1,053,364	44	52,033	72,180	3,779

表1と2より, 単純に局所探索を行った場合と先行関係を用いた局所探索を行った場合とを比べると, 近傍のサイズが単純なものと比較して小さいため, 評価値で比べたときには先行関係を用いた局所探索の方がやや悪くなる傾向がある. しかし, 先行関係を用いた局所探索では計算時間を大幅に削減することができている.

6 おわりに

本研究では, リアルタイムシステムにおけるスケジュールを決定するスケジューリングテーブルという概念を導入し, そのテーブルによって定まるシステムの応答時間が, 設定された許容遅れと呼ばれる時間制約を満たすようなテーブルの探索を行うアルゴリズムを提案した. まず, スケジューリングテーブルの評価を行うために, 有効なパスを全て検出して応答時間を計算する効率的な手続きを提案した. しかし, この評価方法を利用した場合でも, 単純な局所探索法によって近傍解の評価のたびにこの手続を利用すると, 大きな計算時間がかかってしまうことが分かった. そこで, 近傍解の評価が高速に出来るような近傍に限定した探索を行うために, 各タスクの実行に先行関係を持たせ, 近傍解の評価に上述の手続きを要さない局所探索を実現した. その結果, 計算時間の大幅な短縮に成功し, 単純な局所探索では短時間で局所最適解を得ることが困難であるような問題例に対しても, 先行関係を用いた局所探索は高速に動作し, 同じ制限時間のもとで良い解を出力することが確認できた.

参考文献

[1] H. Hashimoto and M. Yagiura. An LP-based algorithm for scheduling preemptive and/or non-preemptive real-time tasks. *Journal of Advanced Mechanical Design, Systems, and Manufacturing*, Vol. 4, pp. 578–587, 2010.

[2] T. Ibaraki and N. Katoh. On-line computation of transitive closures of graphs. *Information Processing Letters*, Vol. 16, pp. 95–97, 1983.

[3] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, Vol. 20, pp. 46–61, 1973.

[4] 村上尚彦, 冨山宏之, 高田広章. 自動車制御分散システムの静的スケジューリング手法. 情報処理学会論文誌, Vol. 48, pp. 203–215, 2007.

[5] 白川洋充, 竹垣盛一. リアルタイムシステムとその応用. 朝倉書店, 2001.